POLYMORPHISM

OBJEKTIF:

1. Mahasiswa mampu memahami mengenai salah satu karakteristik dari Pemrograman Berorientasi Objek yaitu Polymorphism.
2. Mahasiswa mampu memahami konsep Polymorphism yaitu *Dynamic Binding* dan implementasi Polymorphism melalui Inheritance atau Interface.
3. Mahasiswa mampu memahami mengenai materi Class Object pada Java.

5.1 Definisi *Polymorphism*

Polymorphism berarti banyak bentuk. Kata ini merupakan bentukan dua kata: *poly* yang berarti banyak dan *moprh* yang berarti bentuk. *Polymorphism* secara umum adalah penggunaan suatu item baik *interface*, *method*, dan lain-lain pada berbagai macam *object* maupun *entity* yang berbeda-beda dengan syarat suatu *object* atau *entity* tersebut memiliki relasi yang menjembatani agar akses ke item tersebut dapat diperoleh.

Dalam Java, *polymorphism* atau kebanyak-ragaman merupakan keadaan ketika sebuah *reference variable* dapat memiliki tipe *object* yang berbeda-beda meskipun memiliki nama yang sama. *Reference variable* ini memiliki sifat *polymorphic*. Ini berarti *reference variable* mempunyai banyak bentuk. Maksud banyak bentuk dari *reference variable* adalah *reference variable* yang dideklarasikan dengan tipe suatu *class* selain dapat digunakan untuk mereferensikan sebuah *object* dari *class* itu sendiri juga dapat digunakan untuk mereferensikan *object-object* lain dari *subclass-subclassnya*.

Terdapat dua macam *polymorphism* pada Java, yaitu :

1. *Static Polymorphism* (Polimorfisme statis)
2. *Dynamic Polymorphism* (Polimorfisme dinamis)

Static Polymorphism adalah *Polymorphism* yang dilakukan pada waktu *compile* (*compile time*), sedangkan *Dynamic Polymorphism* adalah *Polymorphism* yang dilakukan pada waktu berjalannya program (*runtime*). *Static Polymorphism* bekerja lebih cepat, namun membutuhkan bantuan *compiler* tambahan. Sedangkan *Dynamic Polymorphism* lebih *flexible* contohnya dimana *object* dioperasikan tanpa mengetahui secara penuh tipe dari *object* tersebut, namun *Dynamic Polymorphism* bekerja lebih lambat dari pada *Static Polymorphism*.

Perbedaan dari kedua macam *polymorphism* tersebut terletak pada cara membuat polimorfismenya. *Static Polymorphism* menggunakan *overloading method* sedangkan *dynamic polymorphism* menggunakan *overriding method.*

5.1.1 *Static Polymorphism* dengan *Overloading Method*

Static polymorphism adalah bentuk dari *polymorphism* yang dapat diselesaikan pada saat waktu kompilasi program. *Overloading method* terjadi pada sebuah *class* yang memiliki *method* yang sama tetapi memiliki parameter dan tipe data yang berbeda. Tujuan dari *overloading method* yaitu memudahkan penggunaan atau pemanggilan *method* dengan fungsionalitas yang mirip.

Sebagai contoh, kita akan membuat sebuah *class* dengan nama *class* `Cetak`. Pada *class* ini mempunyai method `maxNumber()`. Perhatikan kode program di bawah ini.

Program (Cetak.java)

```
public class Cetak {

    // Method sama namun parameter berbeda
    // Tipe data double
    static double maxNumber(double a, double b) {
        if (a > b) {
            return a;
        }else{
            return b;
        }
    }

    // Method sama, namun parameter berbeda
    // Tipe data int
    static int maxNumber(int a, int b) {
        if (a > b){
            return a;
        }else {
            return b;
        }
    }

    public static void main(String[] argrs) {
        System.out.println(maxNumber(5.5, 7.5));
        System.out.println(maxNumber(10, 20));
    }
}
```

Output Program (Cetak.java)

```
7.5
20
```

Perhatikan kode program di atas. Pada *class* `Cetak` memiliki dua *method* yang sama yaitu `maxNumber()` namun kedua *method* tersebut memiliki parameter dan tipe data yang berbeda, yaitu :

1. *static double maxNumber(double a, double b)*
2. *static int maxNumber(int a, int b)*

Pada *method* yang pertama memiliki parameter dan tipe data *double*, sedangkan pada *method* yang kedua memiliki parameter dan tipe data *int*. Hal ini jelas berbeda, maka inilah yang disebut dengan *overloading method* yang mana merupakan contoh dari *static polymorphism*. Berikut aturan umum pada *static polymorphism*.

1. *Static polymorphism* dapat memutuskan *method* mana yang akan dieksekusi selama waktu kompilasi.
2. *Overloading method* diperlukan dalam *staitic polymorphism*.
3. *Static polymorphism* terjadi di *class* yang sama.
4. Warisan (*inheritance* ataupun *interface*) tidak terlibat dalam *static polymorphism*.

5.1.2 *Dynamic Polymorphism* dengan *Overriding Method*

Dynamic Polymorphism merupakan *Polymorphism* yang dilakukan pada waktu berjalannya program (*runtime*). *Dynamic polymorphism* biasanya terjadi saat kita menggunakan sistem pewarisan (*inheritance*) dan implementasi *interface*. Seperti yang telah dibahas pada Bab sebelumnya, pada sistem pewarisan kita bisa mewariskan atribut dan *method* dari *superclass* ke *subclass*. *Subclass* akan memiliki nama *method* yang sama dengan *superclass* dan *subclass* yang lainnya.

Jadi *static polymorphism* hanya terjadi dalam satu *class* saja. Sedangkan *dynamic polymorphism* terjadi pada saat terdapat hubungan dengan *class* lain seperti *inheritance* ataupun *interface*.

Dynamic polymorphism menggunakan *overriding method* dalam penerapannya. Terdapat beberapa aturan *overriding method*, sebagai berikut.

1. *Overriding method* digunakan ketika *child class* ingin mengubah *method* yang terdapat pada *superclass* dengan menggunakan nama yang sama.
2. *Subclass* hanya dapat dan boleh meng-*override method superclass* satu kali saja. Tidak boleh ada lebih dari satu *method* yang sama pada *class*.
3. Terkait hak akses, setiap *subclass* tidak boleh mempunyai hak akses *overriding method* yang ketat dibandingkan dengan hak akses *method* pada *superclass*.

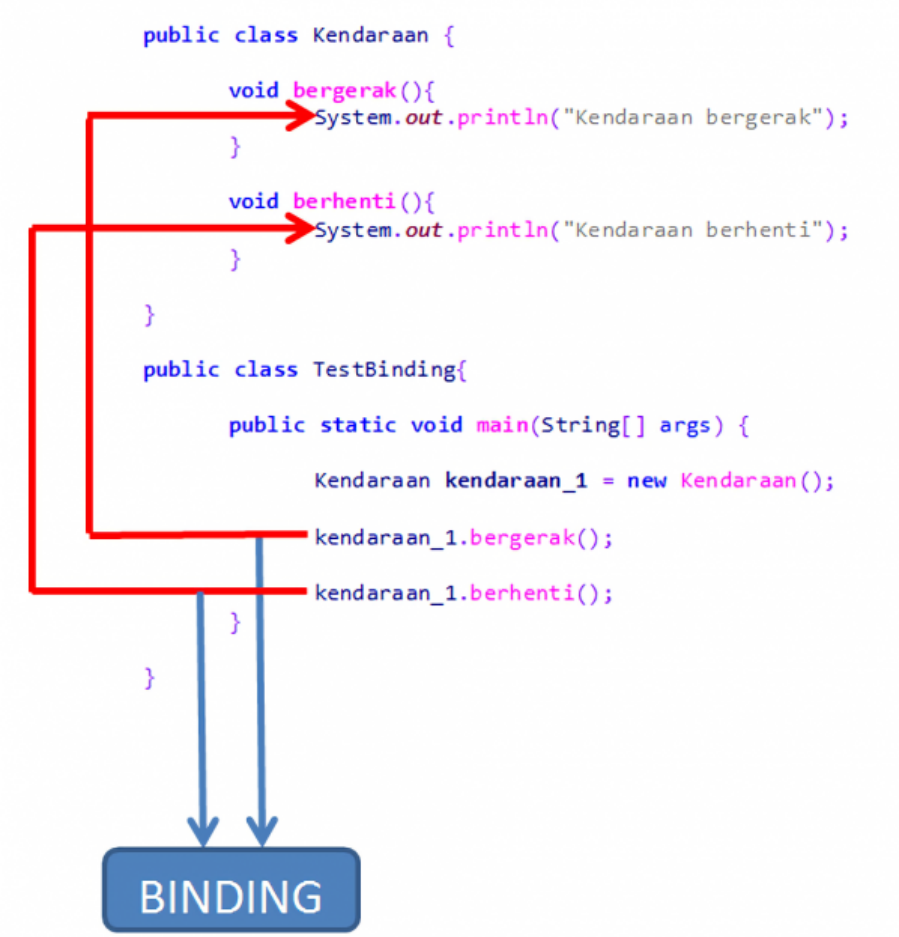
Pada *dynamic polymorphism* terdapat beberapa aturan sebagai berikut.

1. *Dynamic polymorphism* dapat memutuskan *method* mana yang akan dieksekusi dalam *runtime*.
2. *Overriding method* diperlukan dalam *dynamic polymorphism*.
3. *Dynamic Polymorphism* dicapai melalui *dynamic binding*.
4. *Dynamic Polymorphism* terjadi pada *class-class* yang berbeda.
5. *Dynamic Polymorphism* biasanya terjadi pada saat kita menggunakan *inheritance* dan implementasi *interface*.

Untuk materi *dynamic binding*, *polymorphism* via *inheritance*, dan *polymorphism* via *interface* akan dibahas pada sub-bab selanjutnya.

5.2 *Dynamic Binding*

Dalam pemrograman Java pasti tidak terlepas dari yang namanya *method*. *Binding* merupakan hubungan antara pemanggilan *method* dan definisi *method*. Untuk lebih detilnya perhatikan gambar di bawah ini:



Pada gambar di atas, pemanggilan *method* dengan `kendaraan_1.bergerak()` merupakan *binding* yang terhubung dengan definisi *method* `bergerak()` demikian juga dengan `kendaraan_1.berhenti()` merupakan *binding* yang terhubung dengan definisi *method* `berhenti()`.

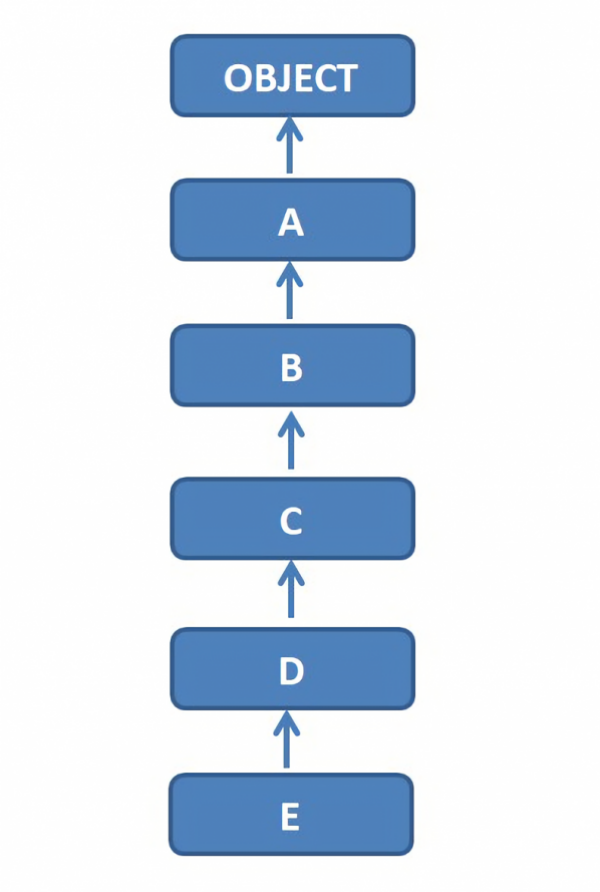
Sebagai aturan pada Java, pada setiap *method* yang dipanggil maka akan selalu terdapat definisi *method*. Karena *error* akan muncul jika *compiler* tidak menemukan definisi *method* yang tepat untuk setiap pemanggilan *method*.

Method dapat diimplementasikan dalam beberapa *class* melalui rantai *inheritance*. Kemudian Java Virtual Machine akan memutuskan mengenai *method* mana yang akan dipilih pada saat *runtime*. Sebelum lebih memahami mengenai *dynamic binding*, terdapat dua istilah yang perlu dimengerti terlebih dahulu, yaitu:

- Tipe yang dideklarasikan, ini adalah *reference variable* untuk *object* atau *declared type*.
- Tipe aktual, yaitu *class* aktual dimana *declared type* menjadi *reference variable* untuk *object* dari *class* tersebut.

Polymorphism atau kebanyak-ragaman merupakan keadaan ketika sebuah *reference variable* dapat memiliki tipe *object* yang berbeda-beda. Keterikatan sebuah *reference variable* untuk menggunakan implementasi *method* dari sebuah *object* inilah disebut *binding*. Ketika keterikatan implementasi sebuah *method* atas sebuah *reference variable* dapat berubah-ubah, maka hal tersebutlah yang disebut dengan *dynamic binding*. *Dynamic binding* merupakan konsep utama dalam penggunaan *polymorphism*. Java menentukan *method* versi mana yang dipanggil dari variabel yang *polymorphic* dengan melakukan *dynamic binding* atau *late binding*. *Dynamic binding (late binding)* berarti penentuan *method* versi mana yang dipanggil dilakukan saat *runtime* (program berjalan).

Mekanisme *dynamic binding* terjadi pada level `run-time` bukan `compile-time`, sehingga *compiler* tidak mengetahui implementasi *method* yang akan dieksekusi sampai program dijalankan. Beikut merupakan contoh dari *dynamic binding*.



- Terdapat beberapa *class* yang terkait dalam rantai *inheritance* yaitu *class* `A`, `B`, `C`, `D` dan `E`, dan paling puncak adalah *class* `Object`.
- Class `E` adalah *subclass* dari *class* `D`, *class* `D` adalah *subclass* dari *class* `C` demikian seterusnya sampai dengan *class* `A` *subclass* dari `Object`.
- Seperti kita ketahui, bahwa *class* `Object` memiliki *method* `toString()`, dan *class* lain bisa meng-*override* *method* tersebut. Dalam hal ini *method* `toString()` tersebut di implementasikan oleh *class* `A`, `B`, dan `C`.

Sering kali tipe data dari *reference variable* sama dengan tipe data *object* yang diinstansiasi, perhatikan contoh program berikut.

```
Object myObject = new A();
```

`myObject` yang berada pada bagian sebelah kiri sebelum operator '`=`', merupakan *reference variable*. Sedangkan `A` yang berada pada bagian sebelah kanan setelah operator '`=`' merupakan *object* yang diinstansiasi. Kesamaan tipe data antara *reference variable* dan *object* tidak mutlak dilakukan, tipe data antara *reference variable* dan *object* harus berhubungan namun tidak harus sama.

Pada contoh di atas, tipe aktual dari *myObject* adalah `A` karena `myObject` mereferensikan suatu *object* yang dibuat dengan `new A();` Selanjutnya, *method* `toString()` manakah yang akan di panggil? Ternyata *method* `toString()` yang akan dipanggil ditetapkan oleh tipe aktual dari `myObject`. Ini di kenal dengan *dynamic binding*. Berikut merupakan contoh *dynamic binding*. Berikut merupakan kode program dari diagram di atas.

Definisi Class(Object.java)

```
public class Object
{
    public String toString()
    {
        return "method toString() dari class Object dipanggil.";
    }
}
```

Definisi Class (A.java)

```
public class A extends Object
{
    // meng-override method toString() yang terdapat pada class Object
    @Override
    public String toString()
    {
        return "method toString() dari class A dipanggil.";
    }
}
```

Definisi Class (B.java)

```
public class B extends A
{
    // meng-override method toString() yang terdapat pada class A
    @Override
    public String toString()
    {
        return "method toString() dari class B dipanggil.";
    }
}
```

Definisi Class (C.java)

```
public class C extends B
{
    // meng-override method toString() yang terdapat pada class B
    @Override
    public String toString()
    {
        return "method toString() dari class C dipanggil.";
    }
}
```

Definisi Class (D.java)

```
public class D extends C
{
    // tidak meng-override method toString()
}
```

Definisi Class (E.java)

```
public class E extends D
{
    // tidak meng-override method toString()
}
```

Program (TestDynamicBinding.java)

```
public class TestDynamicBinding
{
    // method perintah dengan parameter tipe object
    static void perintah(Object myObject)
    {
        System.out.println(myObject.toString());
    }
    public static void main(String[] args)
    {
        perintah(new E()); // memanggil method toString dari class C
        perintah(new D()); // memanggil method toString dari class C
        perintah(new C()); // memanggil method toString dari class C
        perintah(new B()); // memanggil method toString dari class B
        perintah(new A()); // memanggil method toString dari class A
        perintah(new Object());
    }
}
```

Output Program (TestDynamicBinding.java)

```
method toString() dari class C dipanggil.
method toString() dari class C dipanggil.
method toString() dari class C dipanggil.
method toString() dari class B dipanggil.
method toString() dari class A dipanggil.
method toString() dari class Object dipanggil.
```

Pada *method* `perintah()` yang terpadat pada *class* `TestDynamicBinding.java` akan mengambil parameter dengan tipe `Object`, sehingga kita dapat memanggil *method* tersebut dengan *object* apapun, misalnya `new E()`, `new D()`, `new C()`, `new B()`, `new A()`, dan `new Object()`. Ketika *method* `perintah(Object myObject)` dieksekusi, maka argumen *method* `toString()` milik `myObject` akan dipanggil. Yang perlu menjadi perhatian di sini adalah `myObject` ini bisa saja merupakan *instance* dari class `E`, `D`, `C`, `B`, `A` atau `Object`. *Class* `C`, `B`, `A` dan `Object` memiliki implementasi tersendiri pada *method* `toString()` dimana implementasi tersebut akan ditetapkan oleh tipe aktual `myObject` pada saat *runtime*.

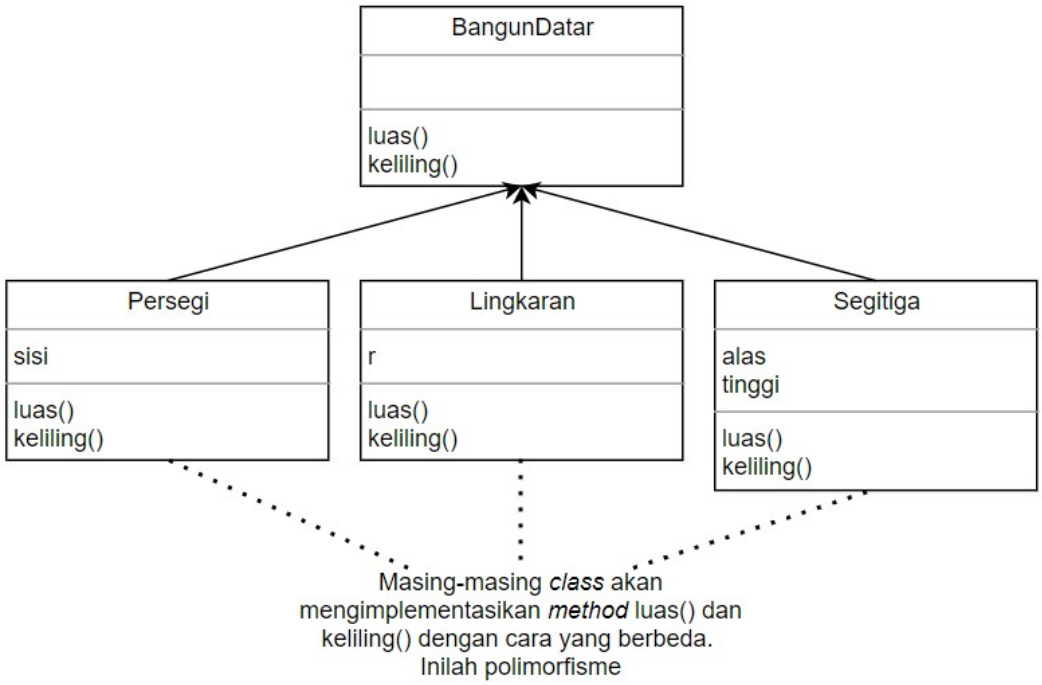
Ketika `myObject` merupakan *instance* dari *class* `E` dan dipanggil pada *method* dengan `perintah(new E());`, maka *compiler* akan mencari *method* `toString()` yang diimplementasikan pada *class* `E`, kemudian jika tidak di temukan akan mencari lagi di *class* `D` dan seterusnya. Karena *class* `E` dan `D` tidak mengimplementasikan *method* `toString()`, maka *compiler* akan menggunakan *method* `toString()` dari *class* `C` ketika `myObject` merupakan *instance* dari *class* `E` atau *class* `D`. Alur pencarian ini akan dilakukan dari *class* yang paling spesifik terlebih dahulu kemudian terus bertahap pada *class* yang paling *general* pada *inheritance*. Ketika `myObject`

merupakan *instance* dari *class* `C`, `B`, `A`, dan `Object`, maka *method* `toString()` yang dipanggil akan sesuai dengan implementasinya masing-masing pada *class-class* tersebut.

Suatu *method* bisa diimplementasikan dalam beberapa *class* melalui *inheritance*. Kemudian Java Virtual Machine secara dinamis melakukan *binding* pada *method* tersebut pada saat *runtime* dan implementasi itu ditetapkan oleh tipe aktual dari variabel. *Dynamic binding* terjadi pada saat *runtime*, dapat disebut juga sebagai *late binding* karena terjadi pada saat program berjalan.

5.3 *Polymorphism* via *Inheritance*

Seperti yang telah dibahas sebelumnya, *dynamic polymorphism* biasanya terjadi saat kita menggunakan sistem pewarisan (*inheritance*) dan implementasi *interface*. Inheritance memungkinkan kita untuk menurunkan/mewariskan atribut dan *method* dari *superclass* ke *subclass*. *Subclass* akan memiliki nama *method* yang sama dengan *superclass* dan *subclass* yang lainnya namun perintah dan parameternya dapat berbeda dengan *superclass* karena *subclass* melakukan *overriding method* yang diwariskannya. Hal inilah terjadi *polimorphism* pada *inheritance*. Sebagai contoh, perhatikan diagram berikut.



Pada diagram di atas, terdapat *class* `BangunDatar` yang memiliki tiga *subclass* yaitu `Persegi`, `Lingkaran`, dan `Segitiga`. Setiap *class* memiliki *method* yang sama yaitu `luas()` dan `keliling()` akan tetapi *method-method* ini memiliki isi rumus yang berbeda. Berikut merupakan contoh kode program berdasarkan diagram di atas.

Definisi Class (BangunDatar.java)

```
public class BangunDatar
{
    public float luas()
    {
        System.out.println("Menghitung luas bangun datar");
        return 0;
    }

    public float keliling()
    {
        System.out.println("Menghitung keliling bangun datar");
```

```
        return 0;
    }
}
```

Definisi Class (Persegi.java)

```
public class Persegi extends BangunDatar
{
    int sisi;

    public Persegi(int sisi)
    {
        this.sisi = sisi;
    }

    @Override
    public float luas()
    {
        return this.sisi * this.sisi;
    }

    @Override
    public float keliling()
    {
        return this.sisi * 4;
    }

    public int getSisi()
    {
        return this.sisi;
    }

    public void setSisi(int sisi)
    {
        this.sisi = sisi;
    }

}
```

Definisi Class (Lingkaran.java)

```
public class Lingkaran extends BangunDatar
{
    int r;

    public Lingkaran(int r)
    {
        this.r = r;
    }

    @Override
    public float luas()
    {
        return (float) (Math.PI * r * r);
    }
```

```
    @Override
    public float keliling()
    {
        return (float) (2 * Math.PI * r);
    }

    public int getR()
    {
        return this.r;
    }

    public void setR(int r)
    {
        this.r = r;
    }
}
```

Definisi Class (Segitiga.java)

```
public class Segitiga extends BangunDatar
{
   int alas;
    int tinggi;

    public Segitiga(int alas, int tinggi)
    {
        this.alas = alas;
        this.tinggi = tinggi;
    }

    @Override
    public float luas()
    {
        return this.alas * this.tinggi;
    }

    public int getAlas()
    {
        return this.alas;
    }

    public void setAlas(int alas)
    {
        this.alas = alas;
    }

    public int getTinggi()
    {
        return this.tinggi;
    }

    public void setTinggi(int tinggi)
    {
        this.tinggi = tinggi;
    }
}
```

Program (TestPolyInheritance.java)

```
public class TestPolyInheritance {
    public static void main(String[] args)
    {
        BangunDatar bangunDatar = new BangunDatar();
        Persegi persegi = new Persegi(4);
        Segitiga segitiga = new Segitiga(6, 3);
        Lingkaran lingkaran = new Lingkaran(50);

        // memanggil method luas dan keliling
        bangunDatar.luas();
        bangunDatar.keliling();

        System.out.println("Luas persegi: " + persegi.luas());
        System.out.println("keliling persegi: " + persegi.keliling());
        System.out.println("Luas segitiga: " + segitiga.luas());
        System.out.println("Luas lingkaran: " + lingkaran.luas());
        System.out.println("keliling lingkaran: " + lingkaran.keliling());
    }
}
```

Output Program (TestPolyInheritance.java)

```
Menghitung luas bangun datar
Menghitung keliling bangun datar
Luas persegi: 16.0
keliling persegi: 16.0
Luas segitiga: 18.0
Luas lingkaran:7853.9814
keliling lingkaran: 314.15927
```

Dapat dilihat berdasarkan kode program di atas, setiap *subclass* yaitu `Persegi`, `Lingkaran`, dan `Segitiga` mengimplementasikan *method* `luas()` dan `keliling()` dengan cara masing-masing. Hal inilah yang dinamakan *polymorphism*.

Ketika sebuah *reference variable* di deklarasikan, *reference variable* tersebut dapat menampung berbagai tipe data *object* selama tipe data *object* tersebut masih berhubungan dengan tipe data *reference variable*-nya. Keterhubungan tipe data dapat dilihat dari struktur *inheritance*. Berdasarkan hirarki *class* sesuai diagram di atas, *class* `BangunDatar` merupakan *superclass* sedangkan *class* `Persegi`, `Lingkaran` dan `Segitiga` merupakan *subclass*. Sehingga dapat dibuat *object* seperti berikut ini:

```
BangunDatar persegi = new Persegi();
```

```
BangunDatar lingkaran = new Lingkaran();
```

```
BangunDatar segitiga = new Segitiga();
```

Pada objek `persegi`, `lingkaran`, dan `segitiga`, menggunakan *reference variable* `BangunDatar`, dengan *object* yang menampungnya adalah `Persegi`, `Lingkaran`, dan `Segitiga`.

Namun, jika pembuatan *object* dilakukan seperti kode di bawah ini, maka tidak dapat dilakukan. Karena antara *reference variable* dan *object* yang menampungnya tidak saling berhubungan.

```
BangunDatar prisma = new Prisma();
```

```
BangunDatar kubus = new Kubus();
```

Polymorphism dapat dilakukan dari *superclass* ke *subclass* atau sebaliknya. Jika *polymorphism* dilakukan dari *superclass* ke *subclass* maka tidak di perlukan *casting* tipe data. Jika *polymorphism* dilakukan dari *subclass* ke *superclass* maka diperlukan *casting* tipe data.

```
BangunDatar persegi = new Persegi();
```

Pada kode program di atas, *polymorphism* dilakukan dari *superclass* ke *subclass*, sehingga tidak diperlukan *casting* tipe data. Sedangkan pada kode program di bawah ini, *polymorphism* dilakukan dari *subclass* ke *superclass*, sehingga dilakukan *casting* tipe data.

```
Persegi bangunDatar = new BangunDatar();
Lingkaran bangunDatar = new BangunDatar();
Segitiga bangunDatar = new BangunDatar();
```

Polymorphism dari *subclass* ke *superclass* secara umum kurang bermanfaat dan sering menimbulkan `run-time` error pada program. Sebisa mungkin hindari *polymorphism* dari *subclass* ke *superclass*. *Polymorphism* tidak bisa dilakukan antar *class* yang tidak memiliki hubungan *inheritance*.

5.3.1 Operator `instanceof`

Terdapat sebuah operator bernama `instanceof` yang dapat kita gunakan untuk menguji apakah sebuah *object* adalah sebuah *instance* dari suatu *class* tertentu. Berikut adalah format umum ekspresi yang menggunakan operator `instanceof`:

```
varRef instanceof NamaClass
```

Ekspresi dengan operator `instanceof` di atas adalah ekspresi Boolean yang mengembalikan `true` jika *object* yang direferensikan oleh `varRef` adalah sebuah *instance* dari `NamaClass`. Jika *object* yang direferensikan oleh `varRef` bukanlah sebuah *instance* dari `NamaClass`, ekspresi ini akan dievaluasi ke `false`.

Sebagai contoh, statement `if` pada kode berikut menguji apakah *reference variable* `bangunDatar` mereferensikan object `BangunDatar`:

```
BangunDatar bangunDatar = new BangunDatar();
if (bangunDatar instanceof BangunDatar)
{
    System.out.println("Ya, bangunDatar adalah BangunDatar.");
}
else
{
    System.out.println("Tidak, bangunDatar bukan BangunDatar.");
}
```

Kode di atas akan menghasilkan *output* "Ya, bangunDatar adalah BangunDatar" karena *object* yang direferensikan oleh variabel `bangunDatar` adalah *instance* dari *class* `BangunDatar`.

Operator `instanceof` juga akan mengembalikan `true` jika *reference variable* diuji terhadap *superclass*nya. Sebagai contoh, perhatikan kode berikut:

```
Persegi bangunDatar = new Persegi();
if (bangunDatar instanceof BangunDatar)
{
    System.out.println("Ya, bangunDatar adalah BangunDatar.");
}
else
{
    System.out.println("Tidak, bangunDatar bukan BangunDatar.");
}
```

Meskipun *object* yang direferensikan oleh `bangunDatar` adalah *object* `Persegi`, kode di atas akan menampilkan "Ya, bangunDatar adalah BangunDatar." Ini karena `Persegi` adalah *subclass* dari `BangunDatar`.

5.3.2 Relasi "Is-a" Tidak Bekerja Dalam Kebalikannya

Perlu dicatat *polymorphism* hanya bekerja jika *reference variable* dideklarasikan dengan tipe *superclass* dan digunakan untuk merefrensikan *subclass*-nya. *Polymorphism* tidak bekerja dalam kebalikannya. Kita tidak dapat mempunyai *reference variable* yang dideklarasikan dengan tipe *subclass* lalu digunakan untuk mereferensikan *superclass*-nya. Sebagai contoh, kode berikut akan menghasilkan error kompilasi:

```
Persegi bangunDatar;
bangunDatar = new BangunDatar();    // ERROR!
```

Pada kode kita mendeklarasikan variabel `bangunDatar` dengan tipe `Persegi`, lalu menugaskan variabel tersebut untuk mereferensikan *object* dari *class* `BangunDatar` yang merupakan *superclass* dari *class* `Persegi`. Kode di atas akan menghasilkan error.

Kode di atas menghasilkan error karena *polymorphism* hanya bekerja satu arah. Alasan kenapa *polymorphism* hanya bekerja satu arah adalah karena *polymorphism* bekerja berdasarkan relasi "is-a". Ingat, *object* dari *class* yang mengekstensi suatu *superclass* mempunyai relasi "is-a" dengan *object superclass*nya. Ini berarti *object* `Persegi` "is-a" (adalah sebuah) *object* `BangunDatar`. Namun, kebalikannya tidak benar. Tidak semua *object* `BangunDatar` adalah *object* `Persegi`.

5.3.3 *Casting*

Seperti yang telah disebutkan sebelumnya bahwa terdapat limitasi pada variabel bertipe data *superclass* yang digunakan untuk mereferensikan *object-object* dari *subclass-subclass*nya. Variabel tersebut hanya dapat memanggil *method* yang dimiliki oleh *superclass*-nya. Bagaimana jika kita ingin memanggil *method-method* yang hanya dimiliki *subclass*-nya? Kita dapat melakukannya dengan terlebih dahulu melakukan *casting* tipe data ke *subclass* tersebut. Sebagai contoh, perhatikan kode berikut:

```
BangunDatar bangunDatar1 = new Persegi();
Persegi bangunDatar2 = (Persegi) bangunDatar1;
```

Pada *statement* kedua dari kode di atas, operator *casting* `(Persegi)` melakukan *casting* variabel `bangunDatar1` yang bertipe data `BangunDatar` ke tipe data `Persegi`. Setelah *statement* ini dieksekusi variabel `bangunDatar2` akan mereferensikan *object* dari *class* `Persegi` yang direferensikan oleh variabel `bangunDatar1`. Dan karena sekarang variabel `bangunDatar2` bertipe data *class* sebenarnya dari *object* yang direferensikan maka variabel `bangunDatar2` mengetahui *method-method* apa saja yang dimiliki oleh *object* tersebut.

Terdapat hal yang perlu diperhatikan saat kita melakukan *casting* tipe *object*: kita harus memastikan *class* tujuan *casting* adalah *class* sebenarnya dari *object* yang ingin di-*casting*. Jika kita melakukan *casting* ke *class* yang bukan *class* dari *object* yang di-*casting*, kita akan mendapatkan *error run-time*. Kita dapat memastikan *class* tujuan *casting* adalah *class* sebenarnya dari *object* yang ingin kita *casting* dengan melakukan pengujian menggunakan operator `instanceof`, seperti dapat dilihat pada kode berikut:

```
if (bangunDatar1 instanceof Persegi)
{
    Persegi bangunDatar2 = (Persegi) bangunDatar1();
    // Sekarang kita dapat memanggil method dari class Persegi
    // melalui variabel bangunDatar2.
}
```

Program berikut, modifikasi dari program `TestPolyInheritance.java` sebelumnya, mendemonstrasikan penggunaan *casting* antar *object*:

Program (TestPolyInheritance2.java)

```
public class TestPolyInheritance2 {
    public static void main(String[] args)
    {
        BangunDatar[] bangunDatar = new BangunDatar[3];

        bangunDatar[0] = new Persegi(4);
        bangunDatar[1] = new Lingkaran(50);
        bangunDatar[2] = new Segitiga(6, 3);

        for (int i = 0; i < bangunDatar.length; i++)
        {
            System.out.println("Luas "+(i+1)+" : " + bangunDatar[i].luas());

            if (bangunDatar[i] instanceof Segitiga)
            {
                Segitiga segitiga = (Segitiga) bangunDatar[i];
                System.out.println("alas segitiga : " +segitiga.getAlas());
                System.out.println("tinggi segitiga : "+segitiga.getTinggi());
            }
            else
            {
                System.out.println("Keliling "+(i+1)+" : "
                                   + bangunDatar[i].keliling());
            }
            System.out.println();
        }
    }
}
```

Output Program (TestPolyInheritance2.java)

```
Luas 1 : 16.0
Keliling 1 : 16.0

Luas 2 : 7853.9814
Keliling 2 : 314.15927

Luas 3 : 18.0
alas segitiga : 6
tinggi segitiga : 3
```

Pada program di atas, di baris 14 sampai dengan 19, kita menuliskan kode yang melakukan *casting* ke tipe data `Segitiga` jika elemen array `bangunDatar` adalah *instance* dari *class* `Segitiga`. Setelah melakukan *casting*, kita memanggil *method* `getAlas()` dan `getTinggi()` yang dimiliki oleh *class* `Segitiga` tetapi tidak dimiliki oleh class `BangunDatar`. Jika `bangunDatar` bukan merupakan *instance* dari *class* `Segitiga`, maka akan dilakukan mencetak nilai keliling dari `bangunDatar`.

5.4 *Class Object*

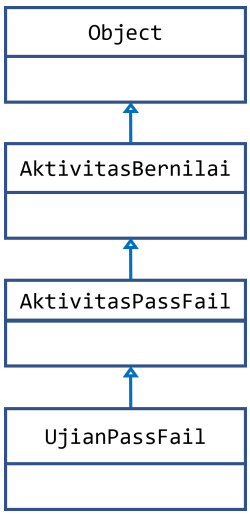
Semua *class* dalam Java secara langsung atau tidak langsung mewarisi dari sebuah *class* bernama `Object`. *Class* yang tidak ditulis dengan *keyword* `extends` untuk mewarisi dari *class* lain, secara otomatis di-ekstensi oleh Java untuk mewarisi dari *class* `Object`. Sebagai contoh, perhatikan definisi *class* berikut:

```
public class MyClass
{
    // Deklarasi Member-member...
}
```

Class di atas tidak secara eksplisit mengekstensi *class* lain, sehingga Java memperlakukan *class* ini seperti seolah *class* tersebut dituliskan seperti berikut:

```
public class MyClass extends Object
{
    // Deklarasi Member-member...
}
```

Semua *class* mewarisi dari *class* `Object`. Sebagai contoh, diagram berikut menunjukkan bagaimana hirarki *inheritance* dari *class* `UjianPassFail` terhadap *class* `Object`:



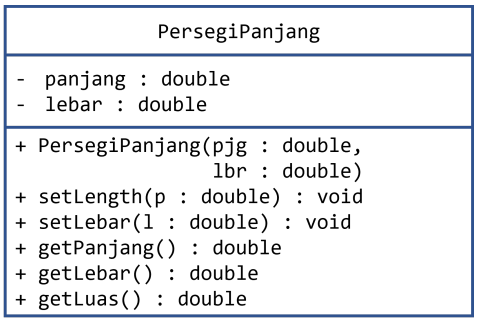
Karena semua *class* secara langsung atau tidak langsung mengekstensi *class* `Object`, maka semua *class* mewarisi member-member dari *class* `Object`. *Class* `Object` mempunyai *method-method* generik, dua diantaranya:

- *Method* `toString`: yang mengembalikan sebuah `String` berisi deksripsi *object*.
- *Method* `equals` : yang membandingkan *object* dengan *object* lain.

Kedua *object* di atas umumnya di-*overriding* oleh *class-class*. *Method* `toString` umumnya di-*overriding* untuk menampikan *state instance* dan *method* `equals` umumnya di-*overriding* untuk membandingkan apakah kedua *object* mempunyai nilai-nilai *field* yang sama.

5.4.1 Meng-overriding Method `toString`

Misalkan terdapat *class* `PersegiPanjang` sebagai berikut.



Kita dapat menuliskan *method* `toString` yang mengembalikan string yang berisi informasi mengenai *state* dari *object* seperti berikut:

Definisi Class (PersegiPanjang.java)

```
public class PersegiPanjang
{
    private double panjang;
    private double lebar;

    public PersegiPanjang()
    {
        panjang = 0.0;
        lebar = 0.0;
    }

    public PersegiPanjang(double panjang, double lebar)
```

```
    {
        this.panjang = panjang;
        this.lebar = lebar;
    }

    public PersegiPanjang(double sisi)
    {
        this(sisi, sisi);
    }

    public void setPanjang(double panjang)
    {
        this.panjang = panjang;
    }

    public void setLebar(double lebar)
    {
        this.lebar = lebar;
    }

    public double getPanjang()
    {
        return panjang;
    }

    public double getLebar()
    {
        return lebar;
    }

    public double getLuas()
    {
        return panjang * lebar;
    }

    public String toString()
    {
        String str = "Panjang: " + panjang +
                     "\nLebar: " + lebar;

        return str;
    }
}
```

Method `toString` dipanggil otomatis ketika kita mengkonkatenasi sebuah string dengan *object*. Sebagai contoh, perhatikan kode berikut:

```
PersegiPanjang boks = new PersegiPanjang(12.0, 5.0);
String s = "Informasi persegi panjang:\n" + boks;
System.out.println(s);
```

Kode di atas akan memberikan *output* berikut:

```
Informasi persegi panjang:
Panjang: 12.0
Lebar: 5.0
```

Method `toString` juga dipanggil otomatis ketika kita memberikan *object* ke *method* yang mencetak ke console (`print`, `println`, atau `printf`). Perhatikan kode berikut:

```
PersegiPanjang boks = new PersegiPanjang(12.0, 5.0);
System.out.println(boks);
```

Kode di atas akan memberikan output berikut:

```
Panjang: 12.0
Lebar: 5.0
```

5.4.2 Meng-overriding Method `equals`

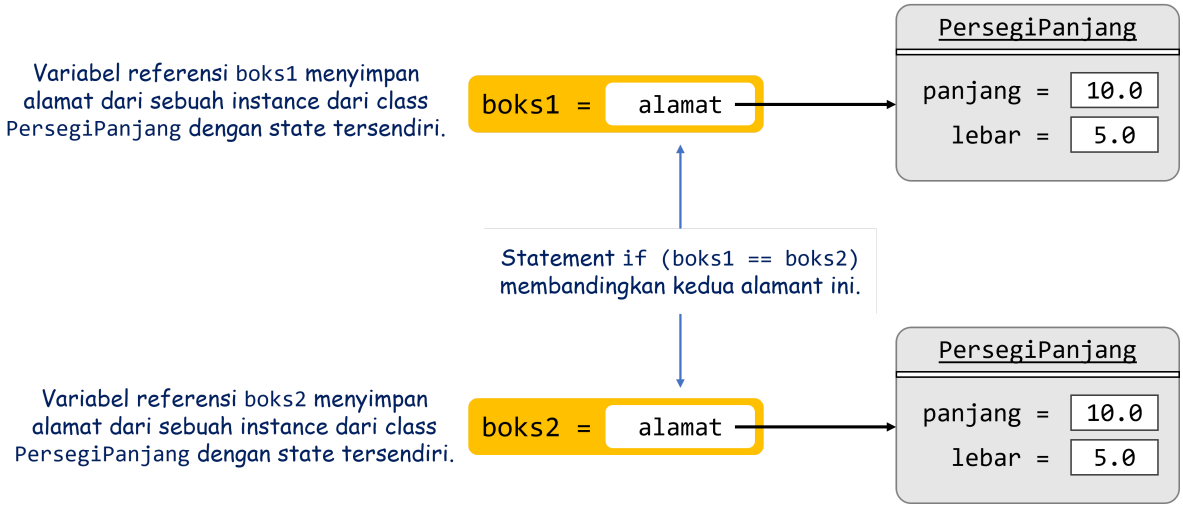
Method `equals` umumnya di-*overriding* untuk menguji apakah *field-field* dari dua *object* mempunyai nilai-nilai yang sama. *Method* ini bekerja berbeda dengan operator `==`. Operator `==` membandingkan apakah dua variabel mereferensikan *object* yang sama, sedangkan *method* `equals` digunakan untuk membandingkan apakah nilai-nilai *field* dari dua *object* adalah sama.

Kita tidak dapat menggunakan operator `==` untuk membandingkan nilai-nilai *field* dari dua *object*. Sebagai contoh, kode berikut terlihat seperti membandingkan isi dari dua *object* `PersegiPanjang`, tetapi sesungguhnya tidak:

```
PersegiPanjang boks1 = new PersegiPanjang(10.0, 5.0);
PersegiPanjang boks2 = new PersegiPanjang(10.0, 5.0);

if (boks1 == boks2)
{
    System.out.println("boks1 dan boks2 memiliki panjang dan lebar yang sama.");
}
else
{
    System.out.println("boks1 dan boks2 memiliki panjang dan "+
                       "lebar yang berbeda.");
}
```

Kode di atas akan memberikan *output*: "boks1 dan boks2 memiliki panjang dan lebar yang berbeda". Ini karena operator `==` tidak membandingkan isi (nilai-nilai *field*) dari *object* `boks1` dan `boks2` tetapi membandingkan alamat memori yang disimpan oleh *object* `boks1` dan `boks2`. Karena `boks1` dan `boks2` mereferensikan dua *object* yang berbeda (meskipun mempunyai nilai-nilai *field* yang sama), maka ekspresi `boks1 == boks2` akan dievaluasi ke `false`.



Kita dapat meng-*overriding method* `equals` pada *class* `Object` sehingga kita dapat membandingkan isi dari dua *object* dari suatu *class*. Kita dapat meng-*overriding method* `equals` pada *class* `PersegiPanjang` dengan menambahkan definisi *method* `equals` seperti berikut:

Potongan Definisi Class (PersegiPanjang.java)

```
public class PersegiPanjang
{
    ...kode-kode yang ditulis pada bagian sebelumnya tidak ditampilkan

    public boolean equals(Object objectLain)
    {
        boolean status;

        // Kita menguji apakah argument object yang diberikan
        // adalah instance dari PersegiPanjang. Pengujian ini untuk
        // memastikan kita dapat melakukan casting.
        if (objectLain instanceof PersegiPanjang)
        {
            PersegiPanjang p = (PersegiPanjang) objectLain;
            if (this.panjang == p.panjang && this.lebar == p.lebar)
            {
                status = true;
            }
            else
            {
                status = false;
            }
        }
        else
        {
            status = false;
        }
        return status;
    }
}
```

Pada kode *method* `equals` di atas kita menggunakan variabel `status` untuk menyimpan nilai Boolean hasil pengujian. Pada baris 12 sampai dengan 27, kita menuliskan *statement* `if` tersarang. *Statement* `if-else` bagian luar menguji apakah `objectLain` yang diberikan sebagai argumen adalah *instance* dari *class* `PersegiPanjang`. Jika `objectLain` yang diberikan bukanlah *instance* dari *class* `PersegiPanjang` maka sudah pasti `objectLain` tersebut tidak mempunyai nilai-nilai *field* yang sama dengan *object* ini. Sehingga kita menetapkan variabel `status` dengan `false` di dalam klausa `else`.

Pengujian apakah `objectLain` adalah *instance* dari *class* `PersegiPanjang` diperlukan untuk memastikan *casting* yang kita lakukan pada baris 14 dapat dilakukan. Jika `objectLain` ini bukanlah *instance* dari *class* `PersegiPanjang` maka *statement* pada baris 14 akan menghasilkan *error*.

Lalu, pada *statement* `if-else` bagian dalam, baris 15 sampai dengan 22, kita menguji apakah *field* `panjang` dan field `lebar` dari *object* ini mempunyai nilai-nilai yang sama dengan *field* `panjang` dan *field* `lebar` dari `objectLain`. Jika ya kita menetapkan `status` dengan `true` dan jika tidak kita menetapkan `status` dengan `false`. Pada *statement* terakhir, baris 28, *method* ini mengembalikan nilai dalam variabel `status`.

5.4.3 Class `PersegiPanjang` dengan Method `toString` dan `equals`

Berikut adalah kode lengkap dari *class* `PersegiPanjang` dengan *method* `toString` dan *method* `equals` yang kita tambahkan sebelumnya:

Definisi Class (PersegiPanjang.java)

```
public class PersegiPanjang
{
    private double panjang;
    private double lebar;

    /*
        Constructor
        @param pjg Panjang dari persegi panjang.
        @param lbr Lebar dari persegi panjang.
    */
    public PersegiPanjang()
    {
        panjang = 0.0;
        lebar = 0.0;
    }

    /*
        Constructor
        @param pjg Panjang dari persegi panjang.
        @param lbr Lebar dari persegi panjang.
    */
    public PersegiPanjang(double panjang, double lebar)
    {
        this.panjang = panjang;
        this.lebar = lebar;
    }

    /*
```

```
        Constructor
        @param sisi Panjang dan lebar dari persegi panjang.
    */
    public PersegiPanjang(double sisi)
    {
        this(sisi, sisi);
    }

    /*
        Method setPanjang menyimpan sebuah nilai dalam field panjang.
        @param pjg Nilai yang disimpan dalam field panjang.
    */
    public void setPanjang(double panjang)
    {
        this.panjang = panjang;
    }

    /*
        Method setLebar menyimpan sebuah nilai dalam field lebar.
        @param lbr Nilai yang disimpan dalam field lebar.
    */
    public void setLebar(double lebar)
    {
        this.lebar = lebar;
    }

    /*
        Method getPanjang mengembalikan panjang dari object PersegiPanjang.
        @return Nilai dalam field panjang
    */
    public double getPanjang()
    {
        return panjang;
    }

    /*
        Method getLebar mengembalikan lebar dari object PersegiPanjang.
        @return Nilai dalam field lebar
    */
    public double getLebar()
    {
        return lebar;
    }

    /*
        Method getLuas mengembalikan luas dari object PersegiPanjang.
        @return Hasil dari panjang kali lebar.
    */
    public double getLuas()
    {
        return panjang * lebar;
    }

    /*
        Method toString mengembalikan sebuah string
        berisi informasi mengenai state dari object.
        @return String deskripsi state object.
    */
```

```
    @Override
    public String toString()
    {
        String str = "Panjang: " + panjang +
                     "\nLebar: " + lebar;

        return str;
    }

    /*
        Method equals mengembalikan Bolean true jika object argument
        memiliki nilai-nilai field yang sama.
        @param objectLain Object yang ingin dibandingkan.
        @return Boolean hasil uji nilai-nilai field.
    */
    @Override
    public boolean equals(Object objectLain)
    {
        boolean status;

        // Kita menguji apakah argument object yang diberikan
        // adalah instance dari PersegiPanjang. Pengujian ini untuk
        // memastikan kita dapat melakukan casting.
        if (objectLain instanceof PersegiPanjang)
        {
            PersegiPanjang p = (PersegiPanjang) objectLain;
            if (this.panjang == p.panjang && this.lebar == p.lebar)
            {
                status = true;
            }
            else
            {
                status = false;
            }
        }
        else
        {
            status = false;
        }
        return status;
    }
}
```

Program berikut mendemonstrasikan *method* `toString` dan `equals` dari *class* `PersegiPanjang`:

Program (DemoPersegiPanjang2.java)

```
public class DemoPersegiPanjang2
{
    public static void main(String[] args)
    {
        PersegiPanjang boks1 = new PersegiPanjang(12.0, 5.0);
        PersegiPanjang boks2 = new PersegiPanjang(12.0, 5.0);
        PersegiPanjang boks3 = new PersegiPanjang(20.0, 12.0);

        System.out.println("Boks1: ");
```

```
        System.out.println(boks1);
        System.out.println();
        System.out.println("Boks2: ");
        System.out.println(boks2);
        System.out.println();
        System.out.println("Boks3: ");
        System.out.println(boks3);
        System.out.println();

        if (boks1.equals(boks2))
        {
            System.out.println("boks1 sama dengan boks2.");
        }
        else
        {
            System.out.println("boks1 tidak sama dengan boks2.");
        }

        if (boks1.equals(boks3))
        {
            System.out.println("boks1 sama dengan boks3.");
        }
        else
        {
            System.out.println("boks1 tidak sama dengan boks3.");
        }
    }
}
```

Output Program (DemoPersegiPanjang2.java)

```
Boks1:
Panjang: 12.0
Lebar: 5.0

Boks2:
Panjang: 12.0
Lebar: 5.0

Boks3:
Panjang: 20.0
Lebar: 12.0

boks1 sama dengan boks2.
boks1 tidak sama dengan boks3.
```

5.5 Polymorphism via Interface

Selain melalui *inheritance*, *polymorphism* juga dapat dilakukan melalui implementasi *interface*. Java memungkinkan kita membuat variabel bertipe data berupa *interface*. Variabel bertipe data *interface* ini dapat digunakan untuk mereferensikan *object-object* yang mengimplementasi *interface* tersebut. Ini adalah *polymorphism* melalui *interface*. Sebagai contoh, misalkan terdapat *interface* bernama `Bentuk` dan terdapat *class* `Lingkaran`, *class* `PersegiPanjang`, dan *class* `Segitiga` yang mengimplementasikan *interface* `Bentuk` sebagai berikut.

Definisi Interface (Bentuk.java)

```
public interface Bentuk
{
    double getLuas();
    double getKeliling();
}
```

Definisi Class (Lingkaran.java)

```
public class Lingkaran implements Bentuk
{
    private double radius;

    /*
        Constructor
        @param radius Radius dari lingkaran.
    */
    public Lingkaran(double radius)
    {
        this.radius = radius;
    }

    /*
        Method getRadius mengembalikan radius lingkaran.
        @return Nilai pada field radius.
    */
    public double getRadius()
    {
        return radius;
    }

    /*
        Method getLuas mengembalikan luas lingkaran.
        @return Luas lingkaran.
    */
    public double getLuas()
    {
        return Math.PI * radius * radius;
    }

    /*
        Method getKeliling mengembalikan keliling lingkaran.
        @return Keliling lingkaran.
    */
    public double getKeliling()
    {
        return 2.0 * Math.PI * radius;
    }
}
```

Definisi Class (PersegiPanjang.java)

```
public class PersegiPanjang implements Bentuk
{
    private double panjang;
```

```
    private double lebar;
    /*
        Constructor
        @param panjang Panjang persegi panjang.
        @param lebar Lebar persegi panjang.
    */
    public PersegiPanjang(double panjang, double lebar)
    {
        this.panjang = panjang;
        this.lebar = lebar;
    }

    /*
        Method getPanjang mengembalikan panjang persegi panjang.
        @return Nilai pada field panjang.
    */
    public double getPanjang()
    {
        return panjang;
    }

    /*
        Method getLebar mengembalikan lebar persegi panjang.
        @return Nilai pada field lebar.
    */
    public double getLebar()
    {
        return lebar;
    }

    /*
        Method getLuas mengembalikan luas persegi panjang.
        @return Luas persegi panjang.
    */
    public double getLuas()
    {
        return panjang * lebar;
    }

    /*
        Method getKeliling mengembalikan keliling persegi panjang.
        @return Keliling persegi panjang.
    */
    public double getKeliling()
    {
        return 2.0 * (panjang + lebar);
    }
}
```

Definisi Class (Segitiga.java)

```
public class Segitiga implements Bentuk
{
    private double a;
    private double b;
    private double c;
```

```
    /*
        Constructor
        @param a Sisi 1.
        @param b Sisi 2.
        @param c Sisi 3.
    */
    public Segitiga(double a, double b, double c)
    {
        this.a = a;
        this.b = b;
        this.c = c;
    }

    /*
        Method getLuas mengembalikan luas segitiga.
        @return Luas segitiga.
    */
    public double getLuas()
    {
        double s = (a + b + c) / 2.0;
        return Math.sqrt(s * (s - a) * (s - b) * (s - c));
    }

    /*
        Method getKeliling mengembalikan keliling segitiga.
        @return Keliling segitiga.
    */
    public double getKeliling()
    {
        return a + b + c;
    }
}
```

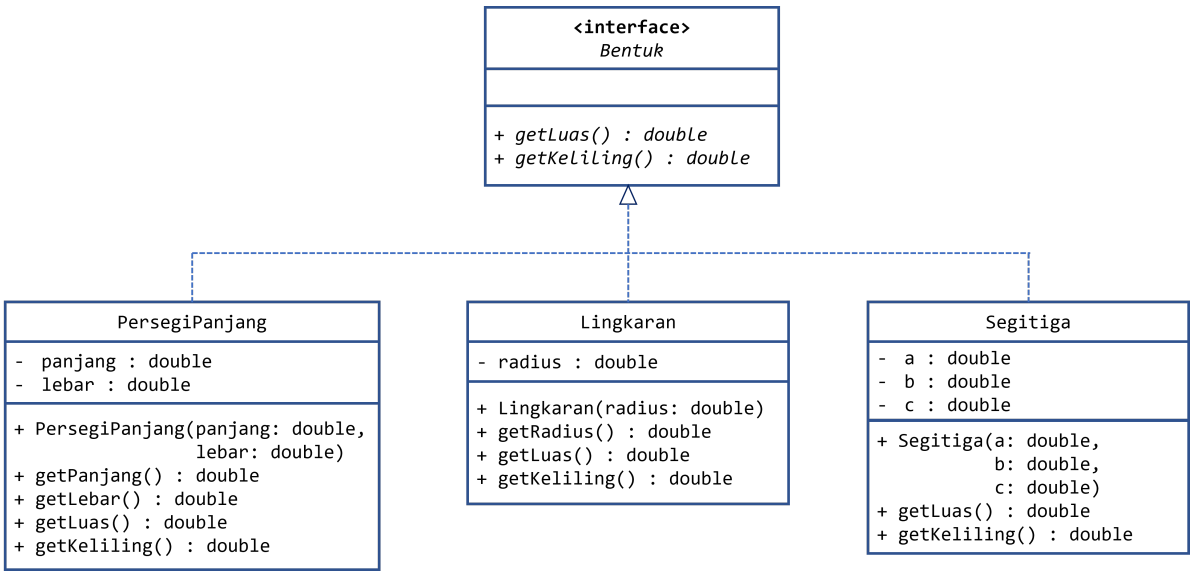
Rumus yang digunakan untuk menghitung luas segitiga pada *method* `getLuas` adalah rumus Heron yang menyebutkan bahwa luas dari segitiga dengan panjang sisi a, b, dan c dapat dihitung dengan:

$$Luas = \sqrt{s(s-a)(s-b)(s-c)}$$

dimana s adalah setengah keliling segitiga, atau

$$s = \frac{a+b+c}{2}$$

Diagram UML dari *interface* `Bentuk` dan class-class yang mengimplementasikannya ditunjukkan oleh gambar berikut:



Keuntungan dari *interface* adalah kita dapat menggunakan *polymorphism* melalui *interface*. Kita dapat mempunyai sebuah array bertipe data `Bentuk` dan menugaskan elemen-elemen pada array tersebut untuk mereferensikan *class-class* yang mengimplementasi *interface* `Bentuk`. Sebagai contoh, program berikut mendemonstrasikan *interface* `Bentuk` dan *class-class* `Lingkaran`, `PersegiPanjang`, dan `Segitiga`.

Program (DemoBentuk.java)

```
public class DemoBentuk
{
    public static void main(String[] args)
    {
        Bentuk[] bangun = new Bentuk[3];
        bangun[0] = new PersegiPanjang(18, 18);
        bangun[1] = new Segitiga(30, 30, 30);
        bangun[2] = new Lingkaran(12);

        for (int i = 0; i < bangun.length; i++)
        {
            System.out.println("Luas = " + bangun[i].getLuas() +
                               ", Keliling = " +
                               bangun[i].getKeliling());
        }
    }
}
```

Output Program (DemoBentuk.java)

```
Luas = 324.0, Keliling = 72.0
Luas = 389.7114317029974, Keliling = 90.0
Luas = 452.3893421169302, Keliling = 75.39822368615503
```

Namun, sama seperti *polymorphism* dengan *inheritance*, *polymorphism* dengan *interface* juga memiliki limitasi: kita hanya dapat memanggil *method-method* yang dirinci pada *interface*. Jika sebuah *class* yang mengimplementasikan *interface* mempunyai *method* lain selain yang dirinci oleh *interface* tersebut, *method* lain ini tidak dapat dipanggil pada variabel bertipe data *interface* yang diimplementasikan *class* tersebut. Misalkan, *class* `PersegiPanjang` mempunyai *method*

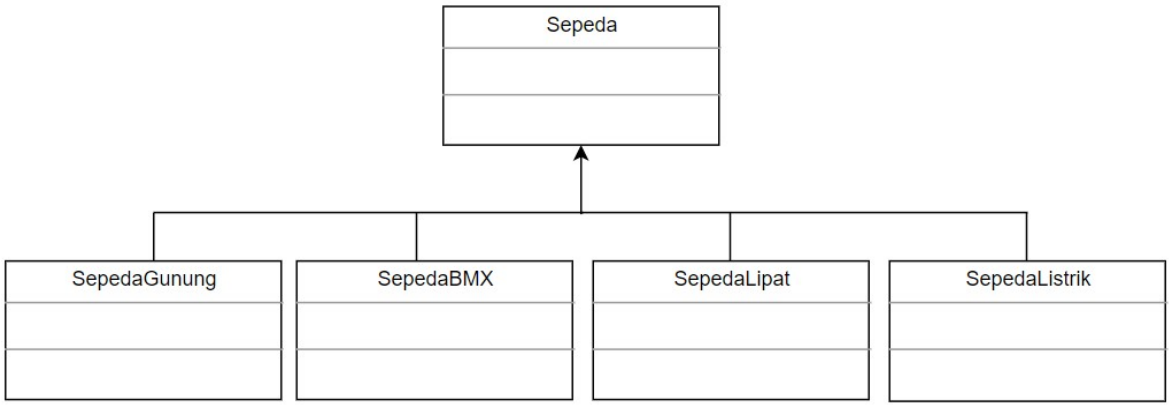
`getPanjang` dan `getLebar`, kedua *method* ini tidak dirinci pada *interface* `Bentuk`, sehingga kedua *method* ini tidak dapat dipanggil melalui elemen array bertipe data `Bentuk`.

Polymorphism pada *interface* tidak dapat dilakukan dengan arah sebaliknya seperti pada *inheritance*, karena *interface* pada dasarnya merupakan *class abstract* dan tidak dapat di instansiasi. *Polymorphism* pada *interface* hanya dapat dilakukan pada *class* yang mengimplementasikan *interface* tersebut.

5.6 Implementasi Polymorphism

Tujuan utama dari konsep *polymorphism* adalah fleksibilitas, sehingga implementasi dari *polymorphism* harus bisa mencapai tujuan tersebut. Pada *polymorphism* fleksibilitas dapat tercapai salah satunya dengan 3 cara yaitu *General Method*, *General Reference Variable* dan *Dynamic Invocation*. *General method* memungkinkan sebuah *method* dapat meng-eksekusi berbagai macam tipe data.

Sebagai contoh, misalkan terdapat *superclass* `Sepeda` dan beberapa *subclass*nya yaitu `SepedaGunung`, `SepedaBMX`, `SepedaLipat`, dan `SepedaListrik`. Berikut diagram dari hirarki *class* tersebut.



Berikut terdapat potongan kode program berdasarkan diagram di atas.

```
SepedaGunung sepeda1 = new SepedaGunung;
SepedaBMX sepeda2 = new SepedaBMX;
SepedaLipat sepeda3 = new SepedaLipat;
SepedaListrik sepeda4 = new SepedaListrik;

sewaSepedaGunung(sepeda1);
sewaSepedaBMX(sepeda2);
sewaSepedaLipat(sepeda3);
sewaSepedaListrik(sepeda4);
```

Pada kode program di atas, terdapat masing-masing *method* `sewa` untuk masing-masing `sepeda`. Jika diimplementasikan ke dalam *polymorphism* dengan *General Method* maka kode program akan menjadi seperti berikut:

```
sewa(sepeda1);
sewa(sepeda2);
sewa(sepeda3);
sewa(sepeda4);
```

Method `sewa()` menjadi *General Method*. *Method* `sewa()` dapat mengeksekusi *reference variabel* `SepedaGunung`, `SepedaBMX`, `SepedaLipat`, dan `SepedaListrik`.

General reference variable memungkinkan sebuah *reference variable* dapat menampung berbagai macam tipe data. Seperti contoh berikut:

```
SepedaGunung sepeda1 = getSepedaGunung();
SepedaBMX sepeda2 = getSepedaBMX();
SepedaLipat sepeda3 = getSepedaLipat();
SepedaListrik sepeda4 = getSepedaListrik();
```

Pada kode program di atas `sepeda1`, `sepeda2`, `sepeda3`, dan `sepeda4` merupakan masing-masing variabel untuk setiap sepeda. Jika diimplementasikan ke dalam *polymorphism* dengan *General reference variable* maka akan menjadi seperti berikut:

```
Sepeda sepeda;
sepeda = getSepedaGunung();
sepeda = getSepedaBMX();
sepeda = getSepedaLipat();
sepeda = getSepedaListrik();
```

`sepeda` merupakan *general reference variable*. Sehingga tidak perlu 4 variable berbeda untuk menampung hasil pencarian `sepeda1`, `sepeda2`, `sepeda3`, dan `sepeda4`. Cukup dengan 1 variable yaitu `sepeda` sudah dapat menampung `sepeda1`, `sepeda2`, `sepeda3`, dan `sepeda4`.

Dynamic invocation memungkinkan sebuah *reference variable* meng-eksekusi(*invoke*) sebuah *method* dengan implementasi yang berbeda-beda, *dynamic invocation* disebut juga *dynamic binding*. Seperti pada contoh berikut:

```
Sepeda sepeda;
sepeda = new SepedaGunung();
System.out.println(sepeda.namaMerk());
sepeda = new SepedaBMX();
System.out.println(sepeda.namaMerk());
sepeda = new SepedaLipat();
System.out.println(sepeda.namaMerk());
sepeda = new SepedaListrik();
System.out.println(sepeda.namaMerk());
```

Pada kode program di atas terdapat *reference variable* yang sama dengan *method* yang sama menghasilkan *output* yang berbeda.

REFERENSI:

[1] Lewis, John, dan William Loftus. 2015. *Java Software Solutions Foundations of Program Design 8th Edition*. London: Pearson Education.

[2] Gaddis, Tony. 2016. *Starting Out with Java: From Control Structures through Objects (6th Edition)*. Boston: Pearson.